AD-A037 128 BROWN UNIV PROVIDENCE R I F/G 5/7

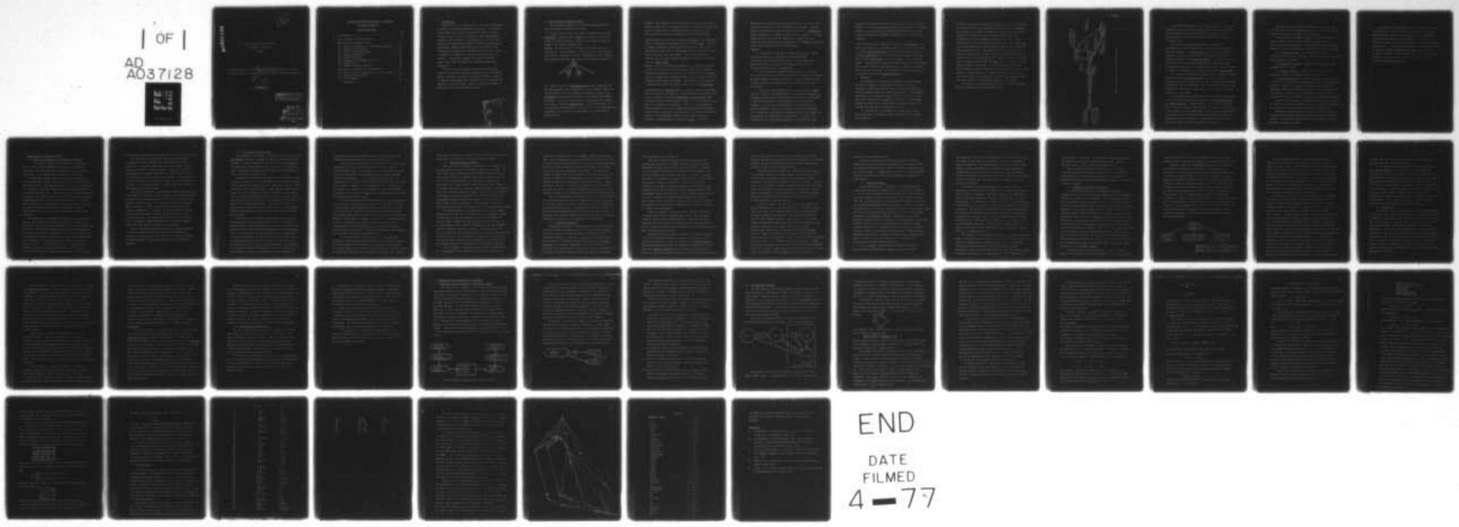
APPLICATIONS OF PATTERN THEORY TO PROBLEMS IN COMPUTER SCIENCE.(U)

OCT 76 N00014-75-C-0461

UNCLASSIFIED NL

| OF |

AD A037128



END

DATE FILMED

4-77

(12)

(6)

Applications of Pattern Theory to Problems in Computer Science

(15)

A Progress Report on Contract N00014-75-C-0461 between the Office of Naval Research, Information Systems Program, and Brown University.

(12) 48p.

(11) ~~October~~ 1976

065 250

Applications of Pattern Theory to Problems in Computer Science

Table of Contents

1. Introduction

Pattern theory is concerned with the study of regularity of objects in particular domains of discourse. Each domain of discourse has its own characteristic kinds of objects. However, it is convenient to develop a domain-independent framework which provides a starting point for the study of regularity in particular domains of discourse. The domain-independent framework described below (in section 2) was developed by Grenander [G1] and applied by him to the study of patterns in mathematics, physics, geology and the life sciences. The present research will apply Grenander's model to the study of problems in computer science and linguistics.

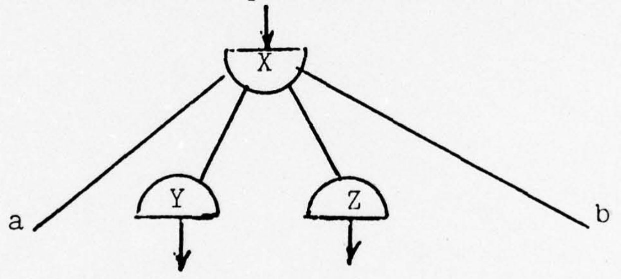
Section 2 of this proposal describes the principal features of Grenander's domain-independent model for pattern theory. Section 3 indicates how certain important concepts in software engineering can be characterized in terms of the concepts of pattern theory. Section 4 proposes research on problems in two specific problem domains.

2. Basic Concepts of Pattern Theory

The principal features of the domain-independent framework for pattern theory are as follows:

(i) Patterns are built up from simple building blocks called generators. Generators have bonds which allow them to be connected to other generators. There are two kinds of bonds referred to as input and output bonds.

Example: The generators of a context free grammar are the productions. The production X → aYZb with three non-terminals X,Y,Z and two terminals a,b may be represented by the following tree structure with one input bond X and two output bonds Y,Z.



(ii) There is a set of combinatory rules which constrains the way that generators may be combined to form composite structures. The combinatory rules are generally expressed in terms of restrictions on bond values at points of interconnection between generators. An object constructed from generators by combinatory rules is called a configuration. We are interested both in individual configurations and in the set of all configurations generatable from a given set of generators by a given set of binatory rules.

Example: The combinatory rules for context free grammars specify that an output bond of a generator can be connected to (identified with) an input bond of a second generator if and only if the bond values are identical (represent the same non terminal).

(iii) Configurations are not necessarily observable. The observable objects determined by configurations are called images. The set of images determines an equivalence relation over the set of configurations such that two configurations are equivalent if and only if they have the same image. A set of configurations together with an image-inducing equivalence relation is referred to as an image algebra.

Example: The set of images determined by a context free grammar is the set of strings of the associated context-free language. Unambiguous grammars have a one to one relation between configurations and images while ambiguous grammars have a many one relation between configurations and images. The problem of reconstructing a configuration given the image is known as the parsing problem.

(iv) Frequently the observer wishes to identify images which differ only in inessential attributes. Observer-induced equivalence over a set of images is handled by similarity transformations which transform one image into another if and only if the observer wishes them to be regarded as equivalent. The set of all similarity transformations over a set of images usually form a semigroup or a group (such as a group of translations or rotations of geometric objects). An equivalence class of images under a class of similarity transformations is called a pattern. A pattern characterizes essential (invariant) attributes of an image while ignoring attributes that are regarded as inessential. Whereas images are observable, patterns may be unobservable abstractions which correspond to concepts in the mind of an observer. In some applications an image may be regarded as a syntactic object while the associated pattern is regarded as the semantic equivalence class of all images having the same meaning.

Examples:

a) The capital letter "A" is regarded as the same object even when subjected to translation or magnification. The pattern associated with the letter A is an unobservable abstraction which is distinct from images associated with individual instances of its appearance.

b) Plato devoted considerable philosophical attention to the relation between the abstract notion of a table and observable concrete instances (images) of a table. Pattern theory develops a formal framework for talking about philosophical problems of this nature.

c) We shall be concerned with the relation between images which are syntactic representations of programs in a programming language and associated patterns which characterize the semantics of the task performed by the program in some specification language. In this case the equivalence class over the set of images (programs) determined by a pattern (task specification) is so complex that the problem of deciding whether two images are instances of the some pattern is undecidable. Moreover, the impossibility of constructing similarity transformations which completely characterize such equivalence classes can be proved. However, the characterization of task specifications as abstract patterns whose concrete realizations are program seems a very natural one and will be further considered in the body of this proposal.

(v) It is assumed that images are subject to <u>deformations</u>. The process of inducing a true image from an observed deformed image is called <u>image restoration</u>. The space of deformed images is called <u>heteromorphic</u> if it is richer than the space of images and is called <u>automorphic</u> if it is a subspace of the space of images. When images are subject to deformation it is the deformed image rather than the image itself that is directly observable. The process of going from an observable possibly deformed image to a pattern is called <u>pattern recognition</u>.

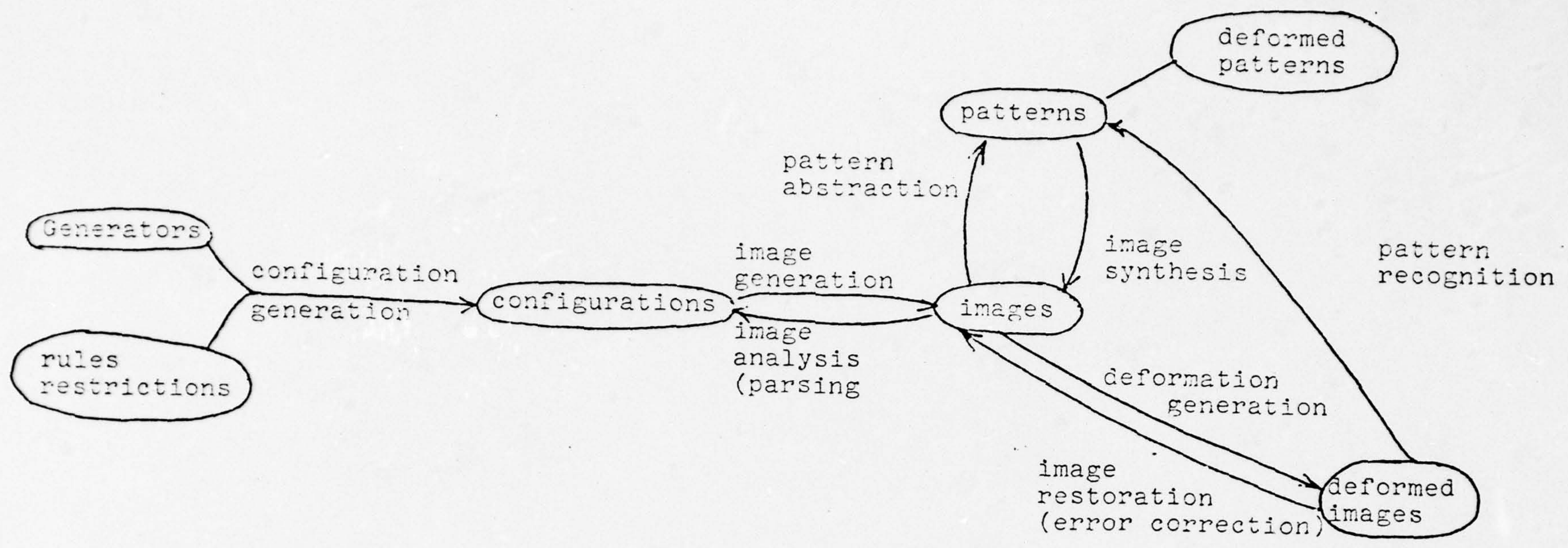
Examples:

a) Messages with errors are examples of deformed images and error correction is an example of image restoration. Automorphic deformation spaces make error detection difficult because deformed images are always valid images. Heteromorphic spaces may be structured so that not only error detection but also error correction is possible. The price of heteromorphic spaces with good error detection and correction properties is considerable syntactic redundancy of messages.

b) Handwriting is an example of an image space whose associated space of deformed images is generally exceedingly heteromorphic.

(vi) There are certain applications where the study of deformations of patterns (rather than deformations of images) is appropriate. For example if we regard a program as an image and the function computed by a program as an associated pattern then it is interesting to study deformations of the function by small modifications of its input-output relation, but less interesting to study deformations of the program by changes in its character string representation. For geometric objects the image is generally continuous and subject to small deformations while the pattern space is generally discrete. However in computer science there are applications where the image space is discrete and the pattern space is continuous. A continuous model of function spaces arising in computer science has been developed by Dana Scott. Example: Program errors in a program which runs correctly for most input values may be regarded as pattern deformations, where the true pattern is the function computed by the program.

The relation between the pattern theory concepts introduced above is summarized by the following figure:



Objects and Activities of Pattern Theory

In the Figure the vertex labels indicate the kinds of objects studied in pattern theory, and edge labels indicate activities which transform objects from one class to another. Brief explanations of each of the activities mentioned in figure 1 are given below.

The process of going from generators and rules to configurations, from configurations to images and from images to deformed images is a generative process, which generates observable structures from primitive building blocks, and is sometimes referred to as pattern synthesis. The inverse process of starting from a possibly deformed image, performing image restoration to obtain a true image, and performing image analysis to obtain a configuration, is sometimes called pattern analysis.

The process of determining a pattern from a possibly deformed image is called pattern recognition. In order to perform successful pattern recognition it it not necessary that the image restoration process reconstruct a true image, but only that image restoration reconstruct an image that is in the same similarity class as the true image.

In considering the application of pattern theory to computer science we shall emphasize the processes of pattern abstraction and image synthesis. In particular we shall consider a domain where images are programs and patterns are program specifications. In this domain, pattern abstraction is the process of determining the specification of a given program (program verification) and image synthesis is the process of developing a program which satisfies a given specification (program synthesis).

The edge between patterns and deformed patterns has been left unlabelled because insufficient work has been done in this area to understand the nature of this activity.

The strength of the model of figure 1 arises from the fact that it embraces within a single model activities that are usually considered separately. Four separate activities can be identified each associated with a characteristic class of models.

a) Syntactic Analysis: Concerned with generation of configurations and images, and with parsing of images to determine their syntactic structure.

b) Deformation Analysis: Study of the relation between deformed and true images.

c) Semantic Analysis: Study of the relation between observable images and conceptually defined patterns.

d) Pattern Recognition: Abstractly pattern recognition can be viewed as a combination of semantic analysis and deformation analysis. However in practice the semantic and deformation components of pattern recognition are not rigidly separated and the result is a hybrid set of concepts and techniques quite distinct from those of semantic analysis and deformation analysis.

Pattern theory provides a framework which allows the study of interactions between generative syntactic processes, deformations, conceptual semantic abstraction and pattern recognition.

Grenander has applied pattern theory to numerous examples in mathematics, physics, geology, neural modelling and the life sciences. The present research aims to apply pattern theory to problems in computer science.

Each domain of application of pattern theory has its own characteristic classes of objects (configurations, images, deformed images, patterns), and requires the development of tools and techniques for manipulating these objects. The objects of computer science are programs and their associated abstractions. In the next section we shall reformulate a number of familiar software engineering problems in terms of the language of pattern theory, and show that such a reformulation suggests new directions of research for some "classical" software engineering problems.

3. Applications to Computer Science

3.1. Syntax and Semantics of Programming Languages

In computer science the objects whose regularity we wish to study include programs, algorithms and programming languages. The specialization of pattern theory to a domain whose images are programs may be developed as follows:

A programming language may be syntactically characterized by a (context-free) grammar specified by a set of generators and combinatory rules. The configurations consist of tree structures in the context free grammar case and more general structures in the non context-free case. Images are obtained from configurations by shedding the superstructure of non terminals (scaffolding) and considering only the associated string of terminals. Deformed images correspond to syntactically incorrect programs. The space of deformed images is sufficiently heteromorphic for error detection and there are even some compilers that perform error correction.

The semantics of a programming language may be defined as a mapping from programs into abstract objects which captures the "essential" attributes of a program while ignoring the "inessential" attributes. The abstract objects which represent the semantics of programs are precisely the patterns of the pattern theory model of programming languages. Different decisions concerning what is to be regarded as essential give rise to different classes of semantic patterns. The most common choice of semantics is to regard the functional input-output behavior of a program is essential and all other attributes as inessential. In this case the "pattern" associated with a program as its functional behavior.

One characteristic that makes this domain particularly interesting is that, although the pattern associated with a program is a conceptual abstraction from the program, it may be partially observed, by program testing. However, the function associated with a program generally has a potentially infinite (or prohibitively large) domain so that complete observation of a function pattern is not possible. Since functions computed by programs are partially observable they have some (but not all) the properties of an image space.

Another characteristic that makes this domain different from other domains is the fact that the semantic attributes which form the basis of the pattern abstraction are related only in a very indirect and subtle way to the observable syntactic features of images (programs). Indeed the relation between semantic and syntactic attributes is so subtle that the equivalence classes over images determined by semantic abstractions have a very complex structure. This complexity is illustrated by the fact that the problem of determining whether two programs (images) compute the same function (pattern) is undecidable.

Many of the interesting and practically important problems of software engineering and the analysis of algorithms are concerned with the study of the structure of equivalence classes of programs (images) realizing a given function specification (pattern).

3.2. The Analysis of Algorithms

The analysis of algorithms is concerned with the analysis of individual algorithms (programs) to determine properties such as average or maximum running time and with the study of classes of algorithms for performing a given task (such as sorting) in order to determine properties of the class of algorithms (such as lower bounds on the running time).

A computational task such as sorting may be thought of as a pattern, and the set of all algorithms or programs for realizing a given task may be thought of as the equivalence class of images associated with the given pattern. Thus from the pattern theory point of view the analysis of specific algorithms is concerned with the study of syntactically defined images in order to determine certain semantic attributes of associated patterns (such as maximum and average running time). The study of classes of algorithms is concerned with the inverse process of determining invariant properties of the set of all images associated with a given pattern.

The interest in the theory of algorithms arises from the domain-dependent fact that equivalence classes of algorithms associated with a given task have a subtle structure whose study requires the use of sophisticated mathematical and combinatorial tools. Knuth [K1] has said that perhaps the most significant discovery generated by the advent of computers will turn out to be that algorithms, as objects of study, will turn out to be extraordinarily rich in interesting properties". The richness of structure of algorithm equivalence classes is convincingly demonstrated in the area of sorting by the large numbers of qualitatively different algorithms that have been proposed and implemented for realizing this relatively simple tasks.

The analysis of algorithms is guaranteed to be an open ended challenge to future computer scientists because of certain undecidability results. For example it is known that the problem of determining whether two arbitrary programs compute the same function is undecidable. An even stronger result asserts the impossibility of finding a set of program transformation primitives which allows a program P_1 to be transformed into a program P_2 if and only if they compute the same function. This second result guarantees the existence of programs for computing the same function which differ in non trivial ways.

A second and perhaps more fundamental source of intractability in the analysis of algorithms arises from the fact that the tasks associated with some programs are difficult to specify. In fact if we require that tasks are defined by a relation between their inputs and outputs, then it can be proved that certain (non halting) programs simply cannot be defined. The set of all tasks (patterns) that can be specified by programs is simply too rich to be characterized by existing formalisms for specifying relations between inputs and outputs. The specification problem for programs will be further discussed in subsection 3.3.

The above discussion does not help us to solve specific problems in the analysis of algorithms but does provide a framework for understanding the methodology of the analysis of algorithms. The present research will explore the methodology of specific analysis of algorithms techniques such as problem reducibility and consider the relation between the analysis of algorithms and the design of algorithms (see discussion of program synthesis below).

3.3. The Specification Problem

The specification problem for programs is the problem of specifying the task which a program is supposed to perform. The most common form of specification for simple programs is by an input-output relation. However only a small proportion of the set of all programs writable in a programming language are specifiable by a simple (short) input-output relation. Many of the larger military and commercial applications have specifications which run into many hundreds of pages. Moreover it can be proved that certain non halting programs (such as operating systems or programming language interpreters) cannot be specified by any input-output relation whatsoever. Such programs can only be specified operationally by a program in a second programming language.

The specification of a problem may be regarded as a what specification of what is to be computed while a program is a how specification of how the computation is to be performed. In certain simple examples the what specification can be given by an input-output relation which is simpler than any how specification for realizing the task. However, as pointed out in the previous paragraph there are many practical commercial and military problems whose what specification is complex, and the set of all how specifications realizable by programs is richer than the set of all what specifications definable by input-output relations.

In the language of pattern theory the "what" specification of a computational task specifys a pattern which captures the important characteristics of all programs (images) realizing the task. Since patterns are in general conceptual abstractions with no direct physical representation, the problem of pattern specification is a critical problem in almost every domain of discourse. In most of the cases considered by Grenander, the equivalence classes of images which are instances of the same pattern may be defined by similarity transformations and pattern elements are defined indirectly as equivalence classes determined by a given similarity relation. In the case of programs it may be proved that the class of all programs functionally equivalent to a given program can never be characterized by a similarity transformation.

One of the objectives of this research is to study the problem of program specification both for simple problems and as sorting and for more complex practical problems which arise in commercial and military applications. In particular we propose to take one specific large specification and analyze the reasons for the complexity of the specification in an effort to develop systematic techniques for reducing the complexity of specifications.

3.4. Program Verification

Program verification is concerned with proving that a given how specification (program) realizes a given what specification. Such proofs require a formal (axiomatic) definition of the programming language, consisting of axioms specifying the input-output behavior of primitive program statements, and a rule of inference for statement composition. The axioms and rule of inference allow input-output relations for a composite program to be proved as theorems using the axioms for primitive statements of the program as a starting point.

The above axiomatic approach has been widely used for proving the correctness of "small" programs but there are some unresolved problems which prevent its being used as a standard tool for program verification in a production environment. One of the problems arises from the fact that, in order to determine the form of a predicate in a loop it is necessary to "creatively" synthesize an inductive hypothesis. However the principal limitations in applying program verification techniques to large programs arise from the fact that large programs generally have unmanageable "what" specification. If the theorem (what specification) to be proved in proving program correctness is several hundred pages long then both the substantive problems in proving the theorem and the problems in showing that the proof is correct are likely to be formidable.

From the viewpoint of pattern theory, program verification is simply the activity of showing that a given image (program) realizes a given pattern (specification). The above discussion indicates that the problem of formulating pattern specifications may be a greater bottleneck than the problem of verifying correctness when the specification is given.

Program verification may be regarded as an ambitious attempt to prove correctness of program execution for <u>all</u> elements of an infinite input domain and may be contrasted with <u>program testing</u>, which establishes correctness for individual elements of the input domain and <u>symbolic execution</u> which establishes correctness for subsets of the input domain associated with particular control paths.

From a pattern theory point of view both program testing and symbolic execution are concerned with the construction of "partial" patterns and with verification that such partial patterns are consistent with the pattern specification.

Program testing is performed by simply running the program for selected values of the input domain, and does not require a formal system for proving theorems about composite program structures from axioms for primitive program statements. However, there is a considerable science associated with picking test data sets which "adequately" test the program. This activity corresponds essentially to finding "good" sets of points in a sample space for verifying that the space as a whole has certain predefined attributes (pattern features).

Symbolic execution requires us to start from axioms for program statements and a rule of inference for statement composition, and allows us to "march forward" from the program start through the program execution tree to determine input output relations for terminal vertices. This process is easier than program verification primarily because inductive hypotheses need never be constructed, but clearly requires a much more sophisticated formal model of program behavior than programs testing. Symbolic execution is intermediate between program testing and program verification both its level of formal sophistication and in its level of pattern reconstruction power. Program testing may be thought of as a process of reconstructing individual points of a pattern, symbolic execution reconstructs subsets (lines, areas) of a pattern, while program verification attempts to reconstruct the complete pattern.

Testing and symbolic execution are important software engineering activities which are examples in a particular domain of activities which can be described in a domain independent way by pattern theory. Although these activities are less demanding formally than program verification, they do not alleviate the specification

3.5. Program Synthesis

Program verification is the process of verifying that a program PROG correctly performs the task specified by a predicate P. In the case of program synthesis we are given a specification P and are required to find a program PROG that correctly performs the task. Program synthesis clearly involves program verification of the synthesized program PROG as a subtask. However the class of programs which can be created by program synthesis is generally a small and relatively well structured subset of the set of all possible programs of a programming language. Systematic (or automatic) program synthesis avoids the necessity of verifying badly structured programs outside the class of synthesizable programs and might actually turn out to be easier than the development of a general purpose verifier for both good and bad programs. However program synthesis does require a problem specification as a starting point and will founder if the specification of the program to be synthesized is several hundred pages long.

In the language of pattern theory program synthesis is concerned with the synthesis of images from a given pattern specification. The set of synthesizable images need contain only one image for each pattern equivalence class and is generally a small relatively well structured subset of the set of all possible images. The verification that a synthesized image realizes an intended pattern may in general be easier then the verification problem for arbitrary images but the pattern specification problem is just as critical for the image synthesis problem as for the verification problem.

The object of program synthesis is to convert a static description P of what is to be computed into a dynamic description PROG of how it is computed. This can be done in a structured way by the stepwise introduction of dynamic features into the static description. At each step one or more statically defined components is decomposed into a sequence of statically and dynamically defined components. This process is called structured program development. The structured development of a program PROG from a specification P consists of a sequence $P_0, P_1, \ldots, P_n$ of successively more dynamic descriptions of P where $P_0=P$, P_n = PROG and P_{i+1} is obtained from P_i by expanding a component of P_i into a more dynamic form. A formal system such as Hoare [H1] may be used to prove for i=0,1,2... that P_{i+1} is "equivalent" to P_i.

The sequence of specifications $P_0, P_1, P_2, \ldots, P_n$ of a structured program development process determine successively smaller equivalence classes of programs with P_0 denoting the class of all programs realizing the specifications P and P_n denoting the single equivalence class containing the program P_n. The set of all possible structured devleopments of a specification P into programs PROG which realize P constitute a lattice whose detailed structure is determined by the precise rules for synthesizing a partial specification P_{i+1} from a partial specification P_i.

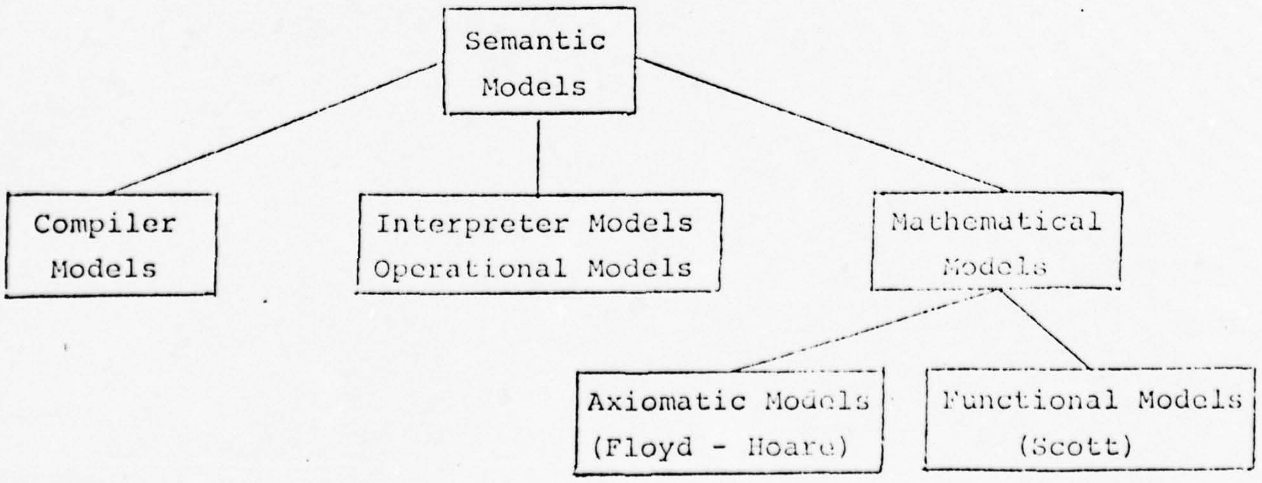
One of the objectives of this research is to examine existing literature on program synthesis with a view to better understanding the limitations which prevent the extension of program verification and program synthesis procedures to realistic classes of programs.

3.6 Semantics and Language Definition

A semantic definition of a programming language L is a mechanism which, given an arbitrary program PROG in the language defines the "meaning" of the program. If the meaning of PROG is the associated task specification P, then a semantic definition supplies P given PROG and may be regarded as an inverse of program synthesis (which supplies PROG given P). Unfortunately not all programs compute functions which are expressible by input output predicates, since the set of "meanings" expressible by input output predicates is restricted to the set of recursive functions while the set of functions computable by programs is the richer set of recursively enumenable functions. The set of input-output predicates P is useful as a basis for specifying many of the tasks that we wish to define in practice but is not an appropriate basis for defining programming language semantics, since a semantic definition of a programming language L should associate a meaning with <u>all</u> programs of a programming language.

A semantic model may be defined as a triple $M = (E,D,\phi)$ where E is a syntactic domain (of programs), D is a semantic domain of denotations and ϕ is a semantic mapping function which maps elements $e \in E$ of the syntactic domain into their denotations $\phi(e) \in 0$.

Semantic models for programming language may be classified in terms of the nature of the domain D of denotations. In particular it is convenient to distinguish between compiler models in which the semantic domain D is a set of programs in a target language, interpreter models in which the meaning of a program is defined in terms of the computations to which it gives rise, and mathematical models in which the meaning of a program is defined in terms of the mathematical function it denotes. Mathematical models may in turn be subdivided into axiomatic models which restrict the semantic domain to total functions and specify functions by a relation between a precondition (inputs) and a post condition (outputs), and functional models (such as those of Scott [S1]) in which the meaning of a program is given by an abstract (partial recursive) function. The relation among these models is given by the following figure:


Semantic Models
Compiler Models
Interpreter Models Operational Models
Mathematical Models
Axiomatic Models (Floyd - Hoare)
Functional Models (Scott)

The above discussion makes it clear that the semantics (meaning) of a program is not an absolute (platonic) notion but rather a relative notion which depends on the context of discourse. When we are concerned with compiling, it is natural to think in terms of a compiler oriented semantics for programs. When we are concerned with the process of execution, it is natural to make use of an interpreter oriented semantics. When we are concerned with program verification, then axiomatic semantics is appropriate. When programs are regarded as abstract mathematical objects then the functional semantics of Scott is appropriate.

Each group of semantic models has given rise to a subculture of computer science with its own group of researchers. The subcultures associated with compiler models, interpreter models and axiomatic models have already been discussed (in the sections on compiler methodology, models of implementation and program verification). The Scott approach is the most abstract and Scott's notion of "meaning" has perhaps a greater claim than any other to be considered _the_ (platonic) meaning of a program. However, one difficulty with Scott's notion of meaning is that the difference between the how specification of a program and the what specification as an abstract function is so great that the mapping from programs to functions cannot be effectively performed. If it could be effectively performed, then we could decide whether two programs realize the same function by mapping them onto their abstract functions and checking for identity. However, we know that the problem of determining whether two programs realize the same function is undecidable (not even partially decidable) and therefore conclude that the semantic mapping function from programs to abstract functions cannot be constructive.

In the language of pattern theory a semantic definition maps images (programs) into the patterns which capture the essential attributes of the images. The patterns which capture the semantics of a given set of observable images are not uniquely defined but are determined by the context of applications. In certain kinds of applications (such as verification and synthesis) it is convenient to consider semantics only for a subset of easily specifiable images and to leave the semantics of certain "intractable" classes of images undefined. However, a complete semantics for a set of images (language) requires a uniform mapping of all images onto their associated patterns.

3.7. Abstraction

An abstraction of an object (program) is a characterization of the object by a subset of its attributes. The attribute subset determines an equivalence class of objects containing the original object as an element. The objects in the equivalence class are called refinements, realizations or implementations of the abstraction. If the attribute subset captures the "essential" attributes of the object then the user need not be concerned with the object itself but only with the abstract attributes. Moreoever if the attribute subset defining the abstraction is substantially simpler than its realizations, then use of the abstraction in place of a realization is substantially simpler than the problem addressed by the user.

The input-output relation realized by a program is an example of a program abstraction. It determines an equivalence class of programs (the set of all programs realizing a given input-output relation). Any program in the equivalence class is a realization (refinement) of the abstraction. The input-output relation captures the essential behavior of the program. When the input-output behavior is a simple or well known mathematical function then use of the abstraction in place of a realization serves a useful purpose. Unfortunately many large commercial industrial and military applications cannot be characterized by functions with a simple input-output behavior, so that the use of abstractions in simplifying the specification of computational problems is strictly limited.

The notion of abstraction is important in the study of program modularity. Modular programming is concerned with breaking a task into modular components where each component has a what specification (abstraction) specifying what the module accomplishes and a how specification (refinement) which specifies how the what specification is realized. If the how specification is specified in terms of a collection of what specifications of lower level modules we are led to stepwise abstraction and stepwise refinement.

In breaking down a program into modular components we frequently encounter "natural" modules consisting of a group of operations acting on a common data structure. For example a stack module may be defined in terms of a group of operations such as push, pop, top, create, testempty. In order to define a stack module we need not specify the internal data structure (linear list, array) used in realizing the module. Moreover we cannot specify individual stack operators such as push by an input output relation because an essential part of their effect is to change the internal data structure. Instead we can characterize the effect of "stack input" operations such as push by axioms such as "top(push(x,stack))=x" which characterize their effect on subsequent "stack output" operations (rush as top) without making any commitment to internal data representations. A specification of stack operations by axioms which are independent of the internal data representation is an example of a data abstraction.

One of the purposes of a module is to erect a "fence" around the how specification of the module which allows systematic information hiding of internal attributes of the module and selective specification of a subset of externally known (exportable) attributes of the module. The associated abstraction must specify the essential characteristics of exportable attributes by some specification mechanism that is independent of the internal how specification of the module. Input-output relations are an appropriate mechanism for procedure abstractions while data abstractions may sometimes be specified by axioms which define the effect of "output operations" in terms of previous input operations. Put our ability to specify abstractions is clearly limited by the fact that many modules we encounter in practice simply do not have simple abstractions.

The above analysis of the notion of abstraction may help us in understanding the goals of modular decomposition of complex programs into modules with clear and hopefully simple abstractions, but the process of decomposing practical programs into modules is still a science rather than an art. One of the goals of this research is to examine how our theoretical understanding of the notion of abstraction can contribute more effectively to practical program development. In order to further this the specifications of modules arising in practical applications will be examined in order to determine the degree to which current intuitively successful techniques are amenable to more rigorous formulation.

3.8. Specific Research Objectives

The above formulation of many of the problems of software engineering in the terminology of pattern theory does not in itself further the state of the art. However, it does provide a framework for understanding the relations among theoretical models and research areas in software engineering which leads to a clearer understanding of the nature of the tools being used and the limitations of current techniques.

The specific research under this project will systematically explore the possibility of applying current theoretical techniques to practical problems by analysis of recent theoretical and practical software technology work from the point of view of pattern theory.

1. An attempt will be made to analyze a number of "classical" papers on program verification, program synthesis, structured programming and specification in order to better understand both the successes and failures of such theoretical papers. One of the products of the work would probably be a survey paper on theoretical models in software engineering.

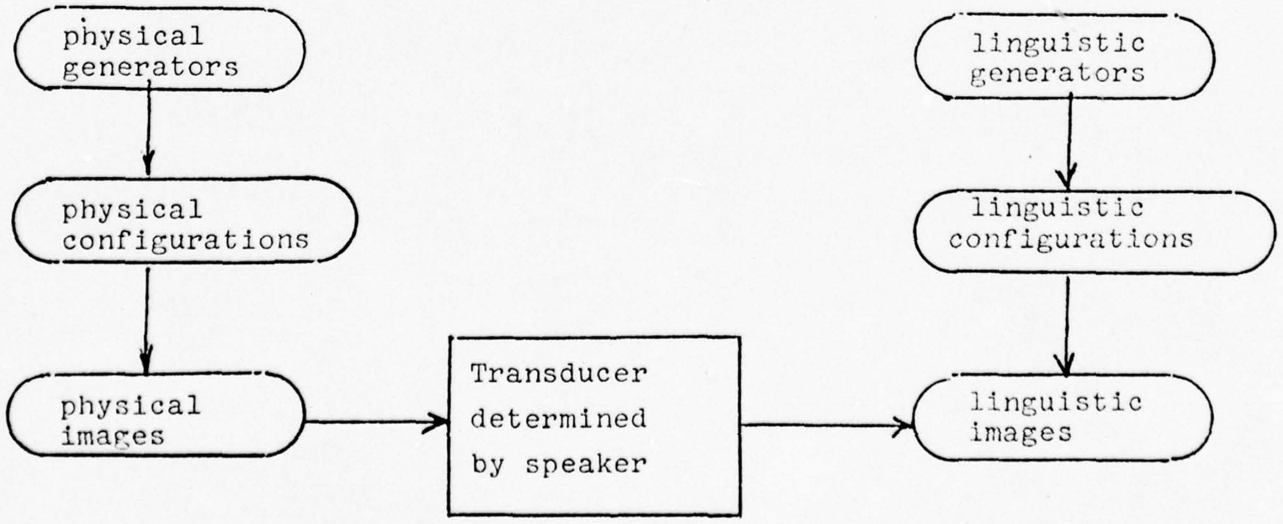
2. A small number of large problem specifications will be examined in detail in order to better understand the sources of complexity in practical program specification and the degree to which such program specifications can be handled by theoretical techniques. The theoretical underpinnings of current software technologies such as structured design will be examined.

It is hoped that the unifying and simplifying view of software engineering that results from viewing it as a branch of pattern theory can be translated into concrete contributions to practical software technology.

4. Application to Linguistics and Abduction.

4.1. Relation between Linguistics and Physical Images.

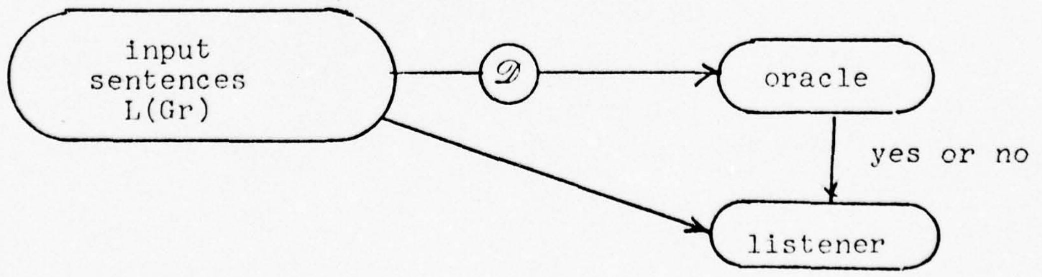
This work will be concerned with the relation between language structure and the objects and relations in the external world which constitute the domain of discourse of the language. From the point of view of pattern theory, sentences of a language are images of an image algebra whose configurations are generated by a phrase structure grammar. The real world objects and relations which give rise to these sentences may be thought of as configurations generated by physical generators which determine an image algebra observed by a speaker (observer). Sentences uttered by the speaker may be thought of as being generated by a "transducer" from the images of the physical world observed by the speaker into linguistic images. The relation between generators, configurations and images of the linguistic and physical worlds is illustrated in the following Figure.



Relation between linguistic and physical images.

We shall consider the case of a listener (which may be a mechanical device) who may observe linguistic images produced by a speaker and has some independent knowledge of the structure of the real world, but does not necessarily have knowledge of the language being used by the speaker. Such a listener may wish to infer the language structure of the language by observing linguistic images, uttered by the speaker. We shall call this problem the abduction problem. Once the language structure is known the listener may wish to "understand" the semantic information about the physical world conveyed by the sentences.

We have made considerable progress in solving the abduction problem for the special case when the language is finite state (see below for detailed discussion). Our technique for inferring the finite state grammar from a sequence of sentences performs systematic deformations on sentences of the language and assumes the existence of an oracle which tells us for each possibly deformed sentence whether it is grammatical or not (see figure).



The technique adopted in our previous work is purely syntactic. In future work we shall investigate the possibility of improving the rate of convergence of the abduction process by making use of semantic information possessed by the listener concerning the structure of the real world.

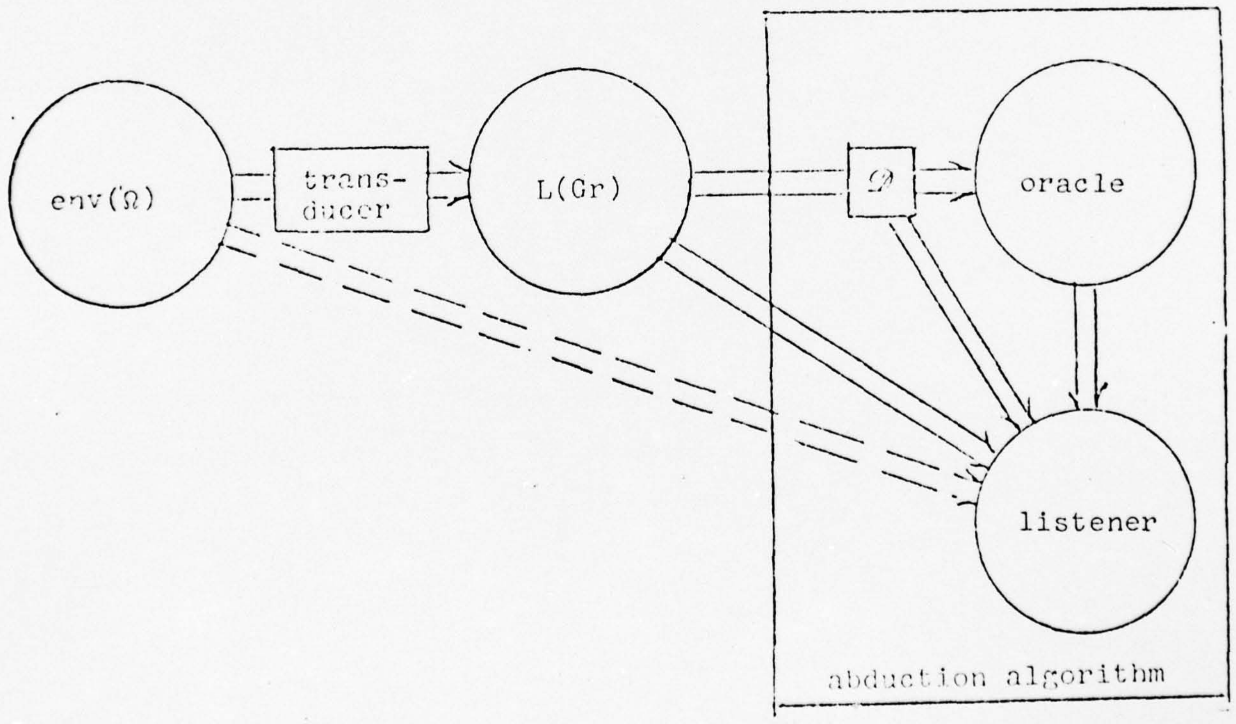
The present research will be concerned with several aspects of the relation between syntactic images in a language and corresponding semantic images of the physical world. Specific problems to be investigated include the following:

1. Investigation of the way in which the learning rate of an abduction process can be speeded up by semantic information.
2. What are the properties of the "transducer" used by the speaker in mapping physical images into linguistic images?
3. What are the limitations imposed by a language or class of languages on the range of objects and relations that can be talked about. There are some specific results in this area concerning formal languages. Are there analogous results for natural languages?
4. How does the structure of the domain of discourse (micro-world) affect the structure of the language for talking about the domain of discourse ?
5. How can we explain the existence of universal language features such as gender, number, active and passive verb forms in terms of our model ?
6. How can other linguistic models such as Chomsky's transformational grammars and Shank's scripts, or Minsky's frames,be expressed in terms of other linguistic models?

4.2 The Abduction Problem.

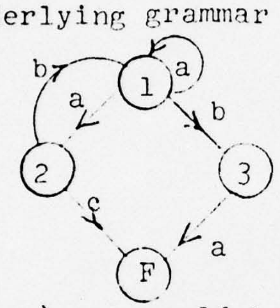
The term abduction was coined by Charles Sanders Pierce to denote the process of generating plausible hypotheses in science. We shall speak of abduction in our context to denote the action of a processor (e.g., a program) on the images from an incompletely known pattern structure in order to generate as output plausible hypotheses concerning the structure. This research project deals, in particular, with the problems which arise when the patterns considered come from some formal language.

The flow of information when we attempt our abduction process can best be described with reference to the following diagram:

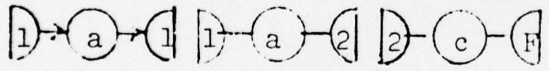


The speaker Ω lives in an environment characterised by some image algebra env(Ω). The generators of this image algebra might be physical objects or, better, physical objects and relations between these objects. A given image belonging to this algebra, I ε env(Ω), can give rise to many syntactically correct sentences belonging to a language L(Gr) described by a grammar Gr and agreeing with I ε env(Ω) semantically: the mapping will be one-to-many.

A grammatical sentence uttered by the observer can be looked upon as an image from another image algebra, with its derivation constituting the corresponding regular configuration. For example, if the underlying grammar is finite state, with the state diagram



the sentence (image) aac would have as one of its possible derivations (regular configurations)



Hence, an image processor maps the image algebra env(Ω) into another image algebra; the image operator "transducer" in the figure takes microworld images into language images.

The sentences from L(Gr) are then subjected to the deformation mechanism $\mathcal{D}$ (the second operator in the figure) which changes an image. In our experiment, this deformation consists of the substitution of words in a sentence (image) by other words or by the substitution of strings for strings. The deformed sentence is presented to the oracle and is also stored in the short term memory of Ω together with the response from the teacher and the undeformed sentences. The oracle will say 'yes' or 'no' according to his judgement of the grammaticality of the sentence: the grammaticality function gr(.), which can take values

YES and NO, is, of course, not known to Ω, and it is important to realise that the oracle may make mistakes. It may judge a sentence grammatical when it is not (with probability δ_1, say), and not grammatical when it is (with probability δ_2, say). The abduction algorithm processes these three inputs and saves the result in the long term memory, after which the short term memory is cleared. As more and more sentences are processed, the algorithm is expected to converge to a limiting grammar weakly equivalent to Gr and with performance parameters characterising a probability distribution over L(Gr) which describes the linguistic performance of Ω.

In addition to the errors $(\delta_1, \delta_2) = \delta$, say, defined above (which were, in fact, neglected in our initial experiments), there is an unavoidable error ε which denotes the probability that two words are placed into the *same* word class when they are, in fact, not equivalent (this will be made clear later). It is important to note that we should wish our algorithm to be robust with respect to the errors δ and ε; our results indicate that in order to achieve robustness w.r.t. ε, abduction algorithms could be realised by network processors (which incorporate parallelism). This robustness (unlike robustness w.r.t. δ) cannot be assured by the serial algorithm we have implemented, and this problem will constitute an important future research project.

We shall describe our abduction algorithm, which assumes that no semantic inputs are available. The incorporation of semantic inputs - indicated in our figure by dotted lines from env(Ω) to listener-will be one of the principal areas of future research.

We displayed two image operators in the figure: the "transducer", which transforms images from env(Ω) into images from L(Gr), and $\mathcal{D}$, the deformation mechanism which in our application consists of word-substitutions and whose outputs are strings over the same terminal vocabulary as that used for inputs into $\mathcal{D}$. This deformation mechanism must be defined in detail by the specifications for the abduction algorithm, and we now address ourselves briefly to this question.

The first question to decide is the type of grammar Gr to use in our studies. For reasons of simplicity we have chosen finite state grammars, but extension to more general models may be possible.
It should be remembered that our goal is not to study abduction of natural languages, but to investigate how abduction can be organised in a concrete environment and in a robust and rapidly convergent manner. For this purpose, the choice of the observer's grammar is not crucial.

Let the terminal vocabulary V_T contain n_T words, denoted generically as x,y,... (or subscripted). The syntactic variables, the non-terminals, form a set V_N with n_N elements, denoted by i,j,... (or subscripted). The re-writing rules (generators)

$$i \rightarrow xj; \quad x \in V_T; \quad i,j \in V_N$$

$$i \rightarrow x \ ; \quad x \in V_T; \quad i \in V_N,$$

the latter resulting in termination of the derivation. The number of re-writing rules is denoted by n_r. The corresponding probabilities are denoted by $p_{ij}(y)$ and $r_i(x)$, forming a matrix P(x) and a vector n(x), respectively. Also,

$$P = \sum_{x \in V_T} P(x)$$

$$r = \sum_{x \in V_T} r(x)$$

For notational and other details about 'linear connection type' configurations, of which the present ones are a special case, probabilities over such configurations, and images corresponding to them, the reader is referred to sections 2.4, 2.10 and 3.10 of reference [G1].

Let x and y be fixed in V_T and pick two arbitrary strings $u,v \in V_T^*$, where V_T^* denotes the (infinite) set of all possible finite strings over V_T. If it is always true that the concatenated strings uxv and uyv are grammatical or ungrammatical together, we say that $x \equiv y$, i.e., they are equivalent (or congruent). In other words

$$x \equiv y \iff gr(uxv) = gr(uyv), \ \forall u,v \in V_T^* .$$

This equivalence partitions V_T into equivalence classes.

Note that equivalence $x \equiv y$ demands that

$$gr(uxv) = gr(uyv)$$

hold for all u,v. Hence an infinite number of tests would be required since V_T^* is infinite. Testing it for a single case (u,v) or a finite number of cases does not suffice, although it is reasonable to feel that if

$$gr(uxv) = gr(uyv)$$

holds for many (u,v)-combinations, then x and y are likely to be equivalent. We shall call

$$\varepsilon = P(\text{decision } x \equiv y \text{ when } x \not\equiv y)$$

where the decision is made after each deformation (substitution). Note that this error is quite different from our previously defined oracle's error

$$\delta = (\delta_1, \delta_2).$$

It can easily be shown that in order that $x \equiv y$ it is necessary and sufficient that for any generator $i \to xj$ there exists one of the form $i \to yj$.

To emphasize the equivalence property we choose as our similarity transformations the set of all permutations of generators (re-writing rules) leaving the equivalence unchanged, so that

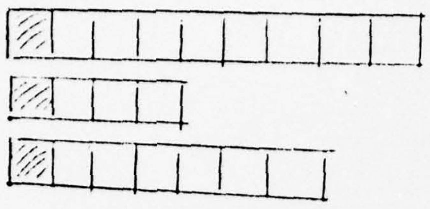
if $\qquad g = i \to xj$

then $\qquad sg = i \to x'j \quad$ with $x \equiv x'$.

This *determines the generator* classes G^{α} left invariant with respect to S. Of course, S is initially unknown and should be learned during the abduction process.

We shall now describe how we determine word classes in one of our abduction algorithms. That is, we shall consider $x, y, \ldots$ to be words rather than elements of V_T^*. The step from word equivalence to string equivalence can be carried out similarly if it is remembered that a state in the state diagram of the grammar can be looked upon as an equivalence class of strings, since, by Nerode's theorem, the number of equivalence classes in V_T^* is finite if the grammar is finite.

Assume that at a given moment during the execution of the algorithm we have arrived at three temporary word classes, which together contain all elements of V_T:



We call the left-hand member of each class the "prototype". Let us say we encounter a sentence

$$x_1x_2x_3 \ldots x_\ell \in L(G);$$

each x_i will occur in one of the lists. Say we select word y for examination:

$$x_1x_2x_3 \ldots y \ldots x_\ell \in L(G)$$

and substitute z for y (deformation $\mathcal{D}$); we now ask whether

$$x_1x_2x_3 \ldots z \ldots x_\ell \in L(G).$$

The oracle will now give the answer YES or NO; we assume the answer to be correct, i.e. $\delta = (\delta_1,\delta_2) = (0,0)$, but the algorithm may, of course, be wrong in placing z and y in the same equivalence class, i.e. we have $\varepsilon \neq 0$. The algorithm proceeds as follows:

if the oracle's answer is NO, y is moved to the class below if there is one, otherwise to a new class placed below the old ones; if the answer is YES, y and z may be equivalent and y is shifted one position to the left. The left most position is called the prototype, and we see that in each temporary class words drifting to the left means their certainty of belonging to this class increases. There are three possible actions for the movement of a word: it may move to the left in the same class, it may move down to one of the existing classes (to the right), it may create a new class. We have shown that the algorithm converges with probability one to a grammar weakly equivalent to the original one, and one can for a given finite state machine, calculate the value of ε or for given words x and y, calculate the corresponding quantity ε_{xy}. An open problem is: given ε, what can we say about the speed of convergence of the algorithm?

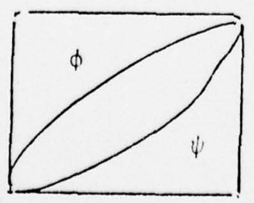
When $\delta \neq 0$, it can be shown that the above algorithm blows up. We have verified this experimentally, but it seems to happen more slowly than one might expect. That is, the algorithm is robust with respect to ε but not with respect to δ. In reference [G2] we presented an algorithm using arrays (parallel processing) rather than lists, which is more space consuming but is robust also with respect to δ. We constructed our incidence matrix

	a	b	c
a			
b	m_{ab}		
c			

where a,b,c are words and m_{ab} is a number indicating our strength of belief in $a \equiv b$. We constructed rules for changing the value m_{ab}:

$$m_{ab} \to \begin{cases} \phi(m_{ab}) & \text{when oracle says YES} \\ \psi(m_{ab}) & \text{when oracle says NO} \end{cases}$$

Clearly, $\psi(m_{ab}) = 0$ when $\delta_2 = 0$, and ϕ and ψ have qualitatively the following form:



Although this incidence matrix algorithm takes a great deal of space, it has to be used if we want robustness with an imperfect oracle. A problem we are currently investigating is the following: say we code a and b into long vectors

$$a \rightarrow y(a)$$

$$b \rightarrow y(b);$$

e.g., y(a) may be the representation of acoustic speech waves of the spoken word a. What happens when we replace the incidence matrix by the corresponding network, where the m's are strengths of connection (amplification or attenuation factors)? What happens when we replace m by ϕ and ψ?

It will be seen from the above that although we have made considerable progress in our analysis of abduction algorithms, several open questions (concerning, for instance, robustness, speed of convergence) still remain, as well as the larger question how semantic inputs can be used to increase speed of convergence to the true grammar of the language.

4.3 A Test Grammar.

To make the above as concrete as possible we constructed a "test grammar". Of course we could have chosen one using a vocabulary consisting of abstract symbols since we are not concerned with natural language processing here. For didactic reasons, however, we have instead selected one generating English-like strings so that the output is easier to read. Since the semantic background has been left out it will be necessary to "fudge" the grammar to avoid completely meaningless sentences from being grammatical.

The grammar has $n_T = 52$ including the punctuation mark "." and a list of the terminal vocabulary is given in Table 1. These 52 "words" are arranged in 23 word classes denoted by, for example, DET for determines, NH for human norm and so on.

	Class Code		Words
1	1	DET	*A
2	1		THE
3	1		SOME
4	2	AJH	*TALL
5	2		CLEVER
6	2		SHORT
7	2		YOUNG
8	3	AJA	*SPOTTED
9	3		FRISKY
10	4	AJ1N	*FINE
11	4		NEW
12	4		VALUABLE
13	5	AJ2N	*BLUE
14	5		ORANGE
15	5		GREEN
16	6	NH	*MAN
17	6		BOY
18	6		WOMAN
19	6		GIRL
20	7	NA	*CAT
21	7		KITTEN
22	7		DOG
23	7		PUPPY
24	8	NN	*TABLE
25	8		CHAIR
26	8		DESK
27	9	AUX	*IS
28	9		WAS
29	10	VP	*SEEN
30	10		HURT
31	10		HELPED
32	11	VT	*LIKES
33	11		DISLIKES
34	12	VI	*SPEAKS
35	12		SINGS
36	13	CONJ	*AND
37	13		WHILE
38	14	PRH	*HE
39	14		SHE
40	15	PRN	*IT
41	16	PNH	*MARY
42	16		JOHN
43	17	PNA	*TOUKA
44	17		ROVER
45	18	ADV	*IMMENSELY
46	18		VIOLENTLY

47	19	VC	*SAYS
48	19	VC	CLAIMS
49	20	REL	*THAT
50	21	BY	*BY
51	22	NOT	*NOT
52	23	DOT	* .

The test grammar Gr has 19 states including the final one F=19 as in the figure. This corresponds to the generators listed in Table 2 , where for example 1 → PNA,7 really represents two re-writing rules since the word class PNA contains two words. In all we have 87 rewriting rules.

A program generates sentences from L(Gr) as described in section 3.2 of ref.[G1]. The performance will of course depend upon the probabilities associated with the generators. Many of the sentences are quite reasonable, such as HE IS HELPED BY A BOY, or JOHN SPEAKS, or THE DESK IS NOT BLUE. Some are a bit doubtful, such as JOHN CLAIMS THAT JOHN SINGS, or SOME VALUABLE TABLE IS NOT GREEN. More seldom one gets a very strange sentence, for example, HE CLAIMS THAT THE MAN CLAIMS THAT A WOMAN IS HELPED BY THE DOG, or SHE VIOLENTLY LIKES THE BOY WHILE HE SPEAKS. It may also be mentioned that some perfectly reasonable looking English sentences over the given terminal vocabulary are not accepted by Gr, for example ROVER LIKES THE GIRL. For our purpose the grammar represents a sufficiently difficult task however.

Consider now the four generators of the form 11 → NH,12. The words in NH are certainly equivalent to each other. Similarly the four generators of type 11 → NA,12 use the words NA which are equivalent to each other. All these eight generators go from state 11 to 12 and one may be tempted to believe that the elements in NA are equivalent to the elements in NH. This is not the case however, since we know that in order that this hold we must have, for example, for the generators 2 → NH,6 generators of the form 2 → NA,6. The latter ones do not appear in Gr, so that

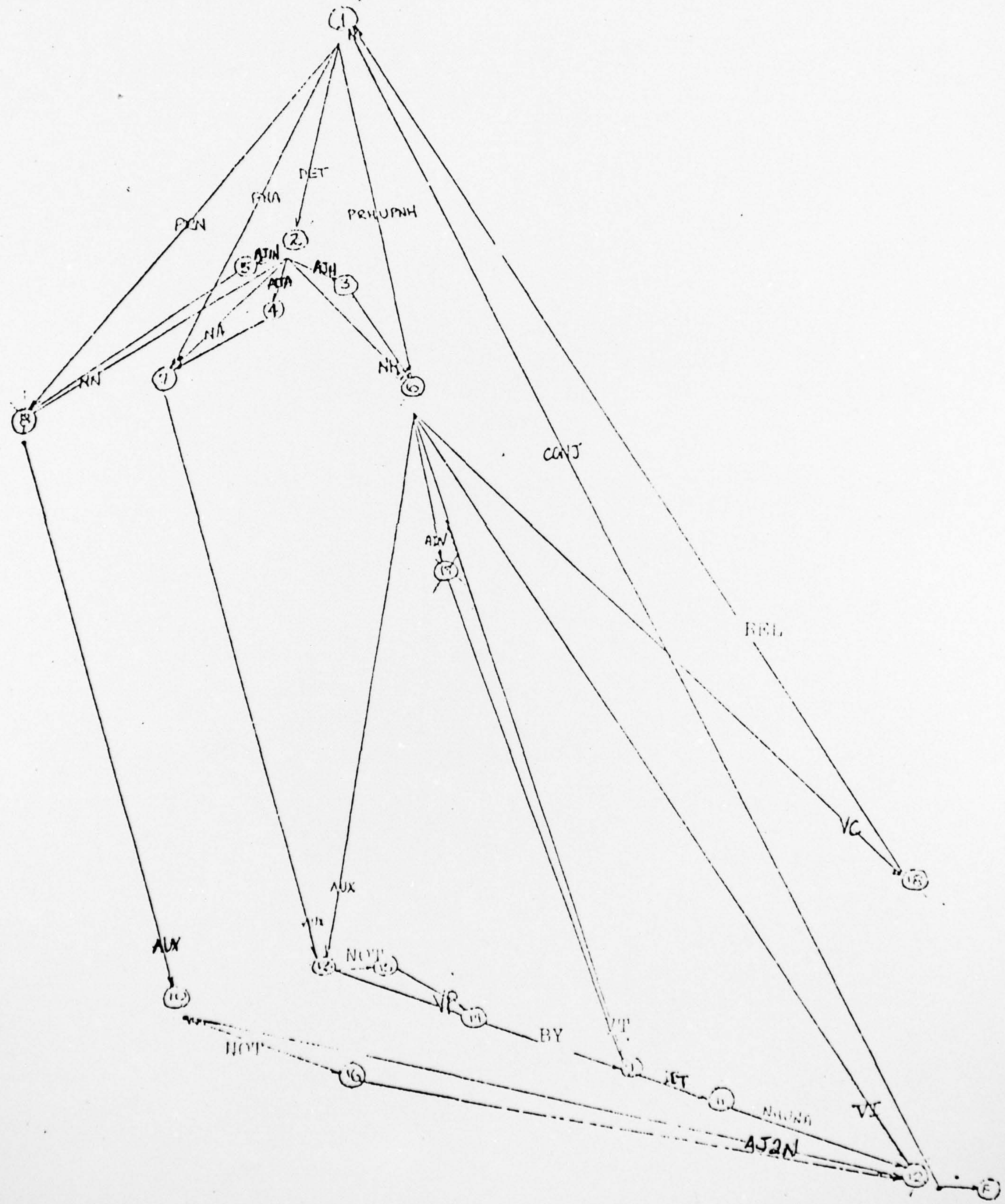

1
DET
PRN
NA
PRNOPNH
2
AJN
AJH
5
ADA
3
4
NA
NN
7
NH
8
6
CONJ
AIN
REL
VC
AUX
AUX
NOT
VP
BY
VT
NOT
DET
NAJNA
VI
AJ2N
E

Table 2

generator number	generator
1	1 → PRN,8
2,3	1 → PNA,7
4,5,6	1 → DET,2
7,8	1 → PRH,6
9,10	1 → PNH,6
11,12,13,14	2 → AJH,3
15,16	2 → AJA,4
17,18,19	2 → AJ1N,5
20,21,22,23	2 → NH,6
24,25,26,27	2 → NA,7
28,29,30	2 → NN,8
31,32,33,34	3 → NH,6
35,36,37,38	4 → NA,7
39,40,41	5 → NN,8
42,43	6 → VT,9
44,45	6 → VI,12
46,47	6 → AUX,13
48,49	6 → ADV,17
50,51	6 → VC,18
52,53	7 → AUX,13
54,55	8 → AUX,10
56,57,58	9 → DET,11
59,60,61	10 → AJ2N,12
62	10 → NOT,16
63,64,65,66	11 → NH,12
67,68,69,70	11 → NA,12
71,72	12 → CONJ,1
73	12 → .,F
74,75,76	13 → VP,14
77	13 → NOT,15
78	14 → BY,9
79,80,81	15 → VP,14
82,83,84	16 → AJ2N,12
85,86	17 → VT,9
87	18 → REL,1

the equivalence between NA and NH does not hold, which will introduce an essential difficulty which we are presently studying.

References

G1. U. Grenander - Pattern Synthesis: Lectures in Pattern Theory, Vol. I, Springer Verlag, 1976.

G2. U. Grenander - Abduction Machines that learn Syntactic Patterns. Division of Applied Mathematics, Brown University Reports on Pattern Analysis, 1976.

S1. Scott, D. and Strachey, S., Towards a Mathematical Semantics for Computer Languages, PRG 6, Oxford University Computer Lab., 1971

H1 Hoare, C.A.R., An Axiomatic Basis for Computer Programming, CACM, October 1969.

K1 Knuth, D. E., Computer Science and its Relation to Mathematics, American Mathematical Monthly, 1973.